INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Update Propagation Strategies to Improve Freshness of Data in Lazy Master Schemes

Esther PACITTI - Eric SIMON

N° 3233

August 1997

THÈME 1

ISSN 0249-6399

Update Propagation Strategies to Improve Freshness of Data in Lazy Master Schemes

Esther PACITTI *- Eric SIMON †

Thème 1 — Réseaux et systèmes
Projet Rodin

Rapport de recherche n ° 3233 — August 1997 — 21 pages

Abstract: Many distributed database applications need to replicate data to improve data availability and query response time. The two-phase-commit protocol guarantees mutual consistency of replicated data but does not provide good performance. Lazy replication has been used as an alternative solution. In this case, mutual consistency is relaxed and the concept of freshness is used to measure the deviation between replica copies. In this paper we present a framework for lazy replication and focus on a special replication scheme called lazy master. In this scheme the common update propagation strategy used is deferred update propagation and works as follows: changes on a primary copy are first commited at the master node, afterwards the secondary copy is updated in a separate transaction at the slave node. We propose strategies based on what we call immediate update propagation. With immediate update propagation, updates to a primary copy are propagated towards a secondary copy as soon as they occur at the master node without waiting for the commitment of the update transaction. We study the behavior of these strategies and show that immediate update propagation may improve freshness with respect to the deferred approach.

Key-words: Distributed Database Systems, Replication, Update Propagation

(Résumé : tsvp)

* The author was supported by Conselho Nacional de Pesquisa (CNPq), Nucleo de Computacao Eletronica of UFRJ (NCE-UFRJ) and is a Phd candidate at PUC-Rio, BRAZIL, e-mail: Esther.Pacitti@inria.fr
† e-mail: Eric.Simon@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Stratégies de Propagation de Mise à Jour pour Améliorer la Fraicheur des Données dans un Schéma de Réplication Asynchrone

Résumé : Les systèmes de gestion de bases de données distribuées ont besoin de répliquer leur données pour améliorer la disponibilité des données et les temps de réponse des requêtes. Le protocole de validation en 2 étapes garantit la cohérence mutuelle des données mais ne fournit pas de bonnes performances. Une alternative à ce protocole est l'utilisation de techniques de réplication asynchrone. Dans ce cas, la cohérence mutuelle n'est plus garantie et le concept de fraicheur sert à mesurer les différences existantes entre les réplicats de données. Nous présentons un nouveau cadre pour la réplication asynchrone et nous nous concentrons sur un schéma de réplication appelé *asynchrone-maître*. Dans ce schéma, la stratégie de propagation des mises à jour utilisée, appelée propagation des mise à jour retardée, fonctionne de la manière suivante: une transaction de mise à jour d'une copie primaire est d'abord validée sur le noeud maître puis la copie secondaire est mise à jour dans une transaction séparée. Nous introduisons de nouvelles stratégies basées sur un principe de mise à jour immédiate. Avec la propagation des mises à jour immédiates, les mises à jour sur une copie primaire sont propagées vers une copie secondaire dès qu'elles sont détectées sur le noeud maitre sans attendre la validation de la transaction de mise à jour. Nous étudions les performances de ces stratégies et nous montrons que la propagation immédiate peut améliorer la fraicheur des réplicats par rapport aux stratégies retardées.

Mots-clé : Bases de données distribuées, Réplication Asynchrone, Propagation de mise á jour

1 Introduction

Data is replicated in database applications to improve query performance and data availability. In a replication scheme, data is replicated through a set of replica copies that are placed on distinct nodes of an interconnected system. The problem is then managing updates over replica copies. With *two-phase-commit* protocol (2PC, each time an update transaction updates its replica copy all other replica copies are also updated inside the same atomic transaction, which guarantees that the replica copies are mutually consistent. However, this solution is limited because replica processing is blocked in the case of network or node failure [PHN87]. In addition performance degrades as the number of nodes increases. An alternative solution relaxes the mutual consistency requirement by using *lazy replication* [Lad90], [Gol95]. In this case, each time an *update transaction* updates its replica copy it is committed locally and afterwards all other replica copies are updated in separate refresh transactions. Since mutual consistency is relaxed, the concept of *freshness* is used to measure the degree of deviation between replica copies [GN95].

The framework we define for a lazy replication scheme is characterized by four basic parameters: owenership, update granularity, refreshment and configuration. The *ownership* parameter [GHOS96] defines the permissions for updating replica copies. If a replica copy is updatable it is called a primary copy otherwise it is called a secondary copy. The *update granularity* parameter defines *when* the updates are to be propagated towards replica copies. Update granularity is said to be *deferred* when update propagation is done after the commitment of an update transaction. In addition, it is said to be *immediate* when update propagation is perfromed after each update on the replica copy. The *refreshment* parameter defines how refresh transactions are managed. Therefore, this parameter is also related to concurrency control protocols. Each combination of an update granularity parameter and refreshment parameter results in a specific *strategy*. The *configuration* parameter is concerned with nodes and network characteristics.

Our work is situated on the *lazy master* replication scheme that is a specific lazy replication scheme. *Lazy master* is used in several replication environments [Dav94]. This scheme fixes the ownership parameter. There is a master node that stores a primary copy and a set of slave nodes that stores secondary copies [GHOS96] of the same data. Our goal is to improve freshness. Therefore, we propose two new strategies that uses what we call *immediate* update granularity called *immediate-immediate* and *immediate-wait*. The difference among the two strategies relies on when refreshment transactions are triggered. With *immediate-immediate* the refresh transaction is triggered for execution when the first update is received. In contrast, with *immediate-wait* the refresh transaction is triggered at commit reception. To understand the behavior of the proposed strategies we perform some experiments and analyze the freshness and query response time results. Furthermore, we compare our strategies to the results obtained by using the *deferred* update granularity, implemented in commercial systems like Sybase. We use as freshness measures the *number of missed versions*. This measure borrows the principle of versions found in multiversions protocols but with respect to a primary copy.

In this paper we show that *immediate* update granularity may improve freshness. For some workloads the improvement is quite significant. However, an increase of queries response times is perceived. We show that by using a multiversion protocol queries response time may be drastically reduced with out a significant loose of freshness.

The remainder of this paper is structured as follows. In Section 2 we present some basic definitions. In Section 3, we present our strategies. In Section 4 we present the freshness measure we use. In addition, we present our performance evaluation and the simulation model with results that shows the trade-offs among the strategies. Section 5 relates our work to other works, section 6 points out future work on the topic. Finally, we conclude in Section 7.

2 Preliminaries

A transaction T consists of a sequence of operations that manipulates a set of database objects. Transactions terminate either successfully by a *commit* operation or unsuccessfully by an *abort* operation in which the ACID (Atomicity, Consistency, Integrity and Durability) properties are assured [PHN87]. Read and write operations executed by T_i are noted generically by r_i or w_i, respectively. Let $T = \{T_1, T_2, ...T_n\}$ be a set of transactions. An *history* is a partial order of the set of operations executed by transactions in T. An history is *serial* if for every two transactions T_i and T_j, either all operations of T_i precede all operations in T_j, or vice-versa. When a transaction contains only read operations it is called *query*. When it contains write operations (update, insert, delete) it is called *update transaction*. The density of an update transaction T_i corresponds to the average interval of time (noted by ϵ) between write operations inside an update transaction. Therefore, the time spent to execute T_i is $(w_1, \epsilon, w_2, \epsilon, ...w_n, \epsilon, commit)$. If, on average, $\epsilon \geq c$, where c is system parameter, T_i is said to be *sparse*. Otherwise, T_i is said to be *dense*.

We assume that the relational model is the common data model used in all nodes and that a replica object is a relation. We use R to denote a primary copy and r denotes a secondary copy of R. Tuples are identified by their primaries keys. Write operations performed by user transactions are logged in a local history log (denoted H) on a stable storage using a Write Ahead Log Protocol [GR92]. This protocol ensures that user transactions operations that are logged and then committed, are subsequently correctly reflected in the database. We assume that each log record has the following structure:

<timestamp, primary_id,tuple_id, item_id, new_value, operation>

Messages are exchanged among the nodes of the replicated system through a reliable communication network that check for errors, loss and duplication of messages. Furthermore, messages are received in the same order of sending (*order preserving*) and fault tolerance is provided by the use of persistent queues.

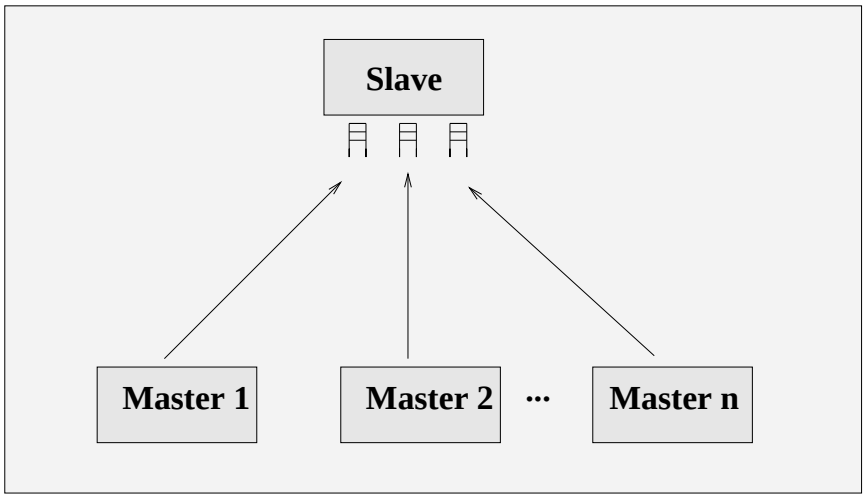

Figure 1: Lazy Master Reception Approach

We focus on the *lazy master replication scheme*. To simplify the description of our strategies and experiments, we consider a *one master-one slave* framework. Each update transaction T_i that updates R has a corresponding refresh transaction, RT_i, that refreshes r. Each refresh transaction RT_i is composed by the sequence of write operations performed by T_i to update R. The process of transmitting these write operations towards the slave is called *update propagation* (for short, we shall e use the term *propagation*). Write operations on primarys copies are captured by reading continuously the log (*log sniffing*) [SKS86, KR87, Moi96].

The slave is exclusively dedicated to query replica copies and its derived data, while the master performs update transactions and local queries. Incomming messges are read from a persistent queue q. For k masters, we assume that the slave node uses one persistent queue q_k to store the incomming messages from each master k (see Figure 1). We call *reception* the process of reading a message or a group of messages from q_k.

3 Strategies

We consider two update propagation granularities: deferred and immediate. For simplicity, we use the terms *deferred update propagation* and *immediate update propagation*. Deferred update propagation is the common approach used in a lazy master replication scheme [Moi96] (see Figure 2). In this case, update propagation occurs after the commitment of each T_i, and the update propagation message contains the RT_i. The log sniffing algorithm used for

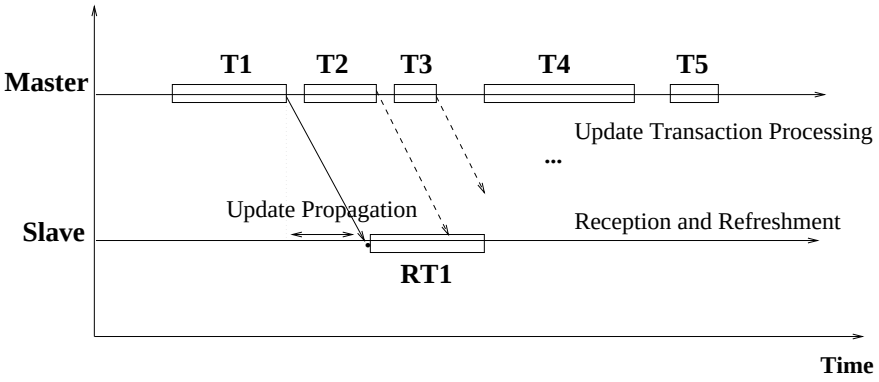

Figure 2: Deferred Update Propagation

```
Monitor(H)
    Repeat
        Search for next begin of T_i ;
        Repeat
            search for w_j;
            read(w_j);
            add w_j to the update propagation message;
        Until (w_j = commit) or (w_j = abort);
        If w_j = commit
            update propagate (RT_i);
    For ever
```

Figure 3: Deferred Log Sniffing

deferred update propagation is shown in Figure 3. Notice that this algorithm guarantees that update propagation is performed using the master serial execution order.

When a slave node receives a RT_i, it is subsequently submitted for execution as shown on Figure 2. Since refresh transactions are executed immediately after their reception, we call this strategy *deferred_immediate*. If T_i is dense, RT_i takes about the same time to execute T_i. Otherwise, if T_i is sparse, RT_i can take less time to execute than T_i.

With an immediate update propagation, update propagation occurs after the execution of each w_j in T_i. Figure shows 5 shows the immediate log sniffing algorithm. Each update propagation message contains a w_j (see Figure 4). It is important to notice that the notion of

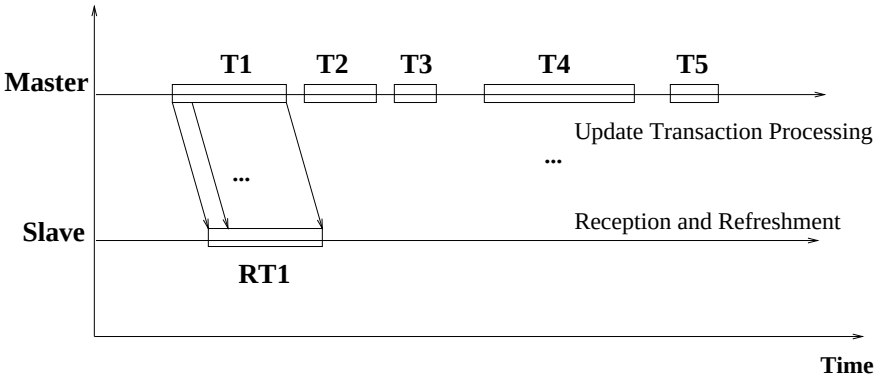

Figure 4: Immediate Update Propagation

```
Monitor(H)
    Repeat
        Search for next begin of T_i;
        Repeat
            search for w_j;
            read(w_j);
            update propagate (w_j);
        Until (w_j = commit) or (w_j = abort);
    For ever;
```

Figure 5: Immediate Log Sniffing

refresh transaction still remains since $(w_1, w_2, ...w_n, commit)$ is executed in the same order. However, the additional network overhead delay introduced to propagate each w_j (noted by δ) may increase update propagation time. Therefore, we consider three refreshment algorithms.

3.1 Immediate_Immediate

With an *immediate_immediate* strategy (see Figure 6), each propagated w_j is read from q_k and subsequently submitted for execution. When *commit* is received the refresh transaction is committed. *Abort* operations that may occur in the master are also propagated and executed in the slave. Concurrent update transactions execution at the master yields

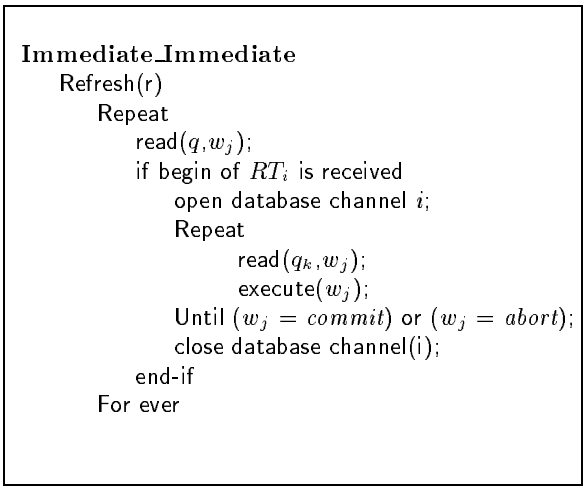

Figure 6: Immediate_Immediate

concurrent refresh transaction at the slave. Therefore, each incomming RT_i is associated with a database connection. Its imortant to notice that the *log sniffing* algorithm guarantees that refresh transactions are executed in update propagation order to follow the master's serial execution order.

Effective refresh time is defined as the delay of time between the commitment of RT_i and its corresponding T_i. When $\epsilon \geq \delta$, effective refresh time corresponds to the time spent to propagate and execute RT_i's *commit*. However, if $\epsilon < \delta$, then effective refresh time may be increased by $n\delta$, where n is the size of T_i. In both cases, replica data items that are being updated will be blocked for reading during a longer period of time, if compared with the deferred approach.

3.2 Immediate_Wait

The second strategy we examine is called *immediate_wait* (see Figure 7). This strategy is an attempt to reduce RT_i execution time. In this case write operations are received one after the other, and the RT_i is triggered for execution only after *commit* reception. When $\epsilon \geq \delta$, effective refresh time corresponds to the time spent to propagate the *commit* plus the execution time of RT_i. Otherwise, if $\epsilon < \delta$, then effective refresh time may be increased by $n\delta$. It is important to notice that network delays do not impact on RT_i execution time, however it may delay the moment at which it is triggered for execution. Futhermore, in case of network or node failures using *immediate_wait* avoids keeping replica copies data items locked during recoverment.

```
Immediate_Wait
    Refresh(r)
        Repeat
            read(q_k,w_j);
            if begin of RT_i is received
                open database channel(i);
                read(q_k,w_j);
                While((w_j ≠ commit) and (w_j ≠ abort))
                        Pre_Compute(RV_i, DV_1, DV_2...DV_n,w_j);
                        read(qk,w_j);
                end-while;
                if w_j = commit
                        apply(RV_i);
                        apply(DV_1, DV_2...DV_n);
                execute(w_j);
                close database connection(i);
            end-if
        For ever.
```

Figure 7: Immediate_Wait Algorithm

Refresh transactions may have different sizes and write patterns. A data item may be written several times inside a refresh transaction. With *immediate_wait*, the result of a sequence of write operations on the same data item may be pre-computed during RT_i reception and applied as a single write operation avoiding redundant disk accesses. For instance, when a sequence of updates on a same data item is received only the last update is applied. Furthermore, materialized derived data refreshment (e.g. aggregates: sum, average) may also be optimized by pre-calculating its final value during reception.

Refresh transaction optimization is done using a dynamic auxiliary structure called *reception vector* (denoted by RV_i), one for each RT_i that refreshes r. This vector has one entry point for each data item being updated. In addition, for each materialized derived data that uses r a *derived data variable* DV_d is also defined to pre-compute its new value. At *commit* reception each entry point of RV_i stores the last pre-computed value for each updated data item. The contents of RV_i is then applied to r. In the same way, the contents of each DV_d is applied to each materialized derived data. The *apply* function reads each entry point of RV_i and each DV_d to update r and the derived data, respectively.

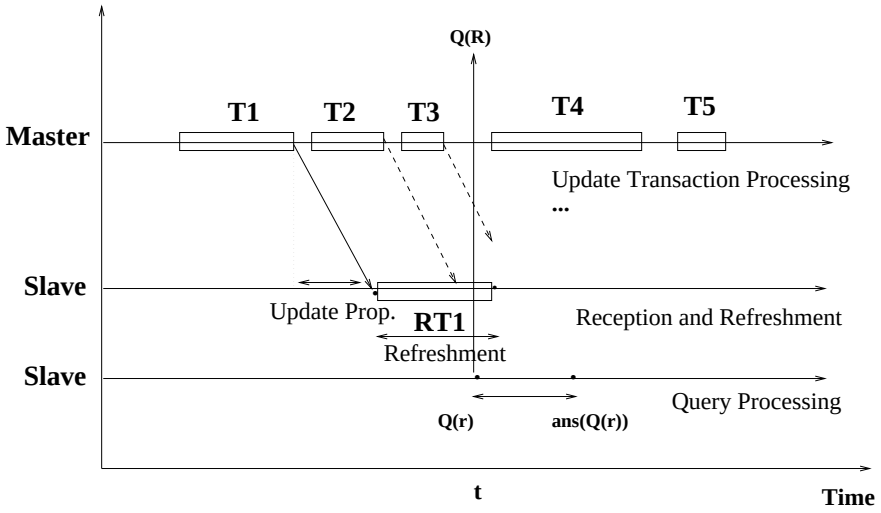

Figure 8: Freshness Measure

3.3 Immediate_Lock

The third strategy is called *immediate_lock*. With *immediate_lock* $(w_1, w_2, ...w_n, commit)$ is received one after the other. However, in this case, each w_j received triggers the locking of the data item it updates. This strategy provides the freshness issues of *immediate_immediate* by blocking the reading of stale data and provides the pre-computations features of *immediate_wait*.

4 Performance Evaluation

4.1 Objectives

In a lazy master replication scheme a deviation between R and r may occur as a result of a network delay, that slows down update propagation, the update transaction workload at the master node, and the query workload at the slave node.

Figure 8 shows a timing graph when using *deferred_immediate* update propagation. Three time lines are shown that corresponds to the serial execution of update transactions at the slave, the reception and execution of RT's and query processing at the master.

The example of figure 8 shows a workload were a deviation between the primary and secondary copy occurs . In this situation, during Q evaluation RT's were in there way to be received or in execution at the slave. Therefore, at time t the state of r does not correspond to the current state of R.

More precisely, consider that the serial execution of $\{T_1, T_2...T_i\}$ generates the respective sequence of states of R, $(R_1, R_2...R_i)$, where each subscript i denotes a version number of R. Similarly, consider that the serial execution of $\{RT_1, RT_2...RT_j\}$ generates the respective sequence of states of r, $(r_1, r_2...r_j)$, where each subscript j denotes a version number of r. Now suppose that at time t, Q is being simultaneously processed at the master and slave nodes. In this situation the version number of R is i' and the version number of r is j'. The *freshness* measure that we use in our experimentations corresponds the *number of missed versions* expressed as $V(R_{i'}, r_{j'}, t) = i' - j'$. For instance, $V(R_5, r_2, 10) = 3$ means that at time 10 there are 3 states of R that were not perceived by a query that reads r.

Freshness may be improved by increasing slave nodes or network speeds. However in our experimentations we study freshnees improvement when using *immediate_immediate* and *immediate_wait* strategies.

4.2 Framework

To perform our practical experimentations we implement a *simulator* shown in Figure 9. The performance model we use is presented in Table 1. Four processes that executes *asynchronously* are defined. The first, *source*, simulates *log sniffing* for a single database connection on the master. Write arrivals in the log is simulated be defining ϵ. We focus in dense update transactions and vary its arrival rate distribution (noted by λ_t). Update transactions are composed by a sequence of *write* operations, each write *updates* a different tuple in the same attribute (noted by *atr*). Each update propagation message is written in order in *pipe* p_1 that is afterwards read by the *update propagator* process.

The *update propagator* process, simulates the communication network. This process has as input the propagation messages written by the *source* process. The network is modeled as a FCFS server. Network delay is calculated by calculating $\delta + t$, where δ is the network delay introduced to propagate each message and t is the transmission time *on-wire*. In general, δ is considered insignificant and t is calculated by dividing the message size by the network bandwidth [CFLS91]. In our experimentations, we fix a short message transmission time (noted by t_{short}) to 300ms. This time corresponds to the time spent to transmit a write message and its size corresponds to a log record. In addition, we consider that the time spent to transmit a RT is linearly proportional to the number of write it carries. The network overhead delay to propagate each message is modeled by the system overhead to read and write from pipes.

With immediate update propagation the propagation time is calculated by $n(\delta + t_{short})$ while with deferred update propagation the propagation time is $(\delta + nt_{short})$. *Network contention* occurs when δ increases due to the increases of traffic in the network. In this situation, the delay introduced by δ may impact update propagation time specially with immediate update propagation. The output of the *update propagator* process are the messages that are written in order in pipe p_2 that are afterwards read by the *refresher* process.

The third process, *refresher*, implements message reception and refreshment at the slave node. Refresh transaction execution is performed on top of Oracle 7.3/Sun-Solaris system using C/SQL. Each write operation corresponds to an UPDATE command that is submitted

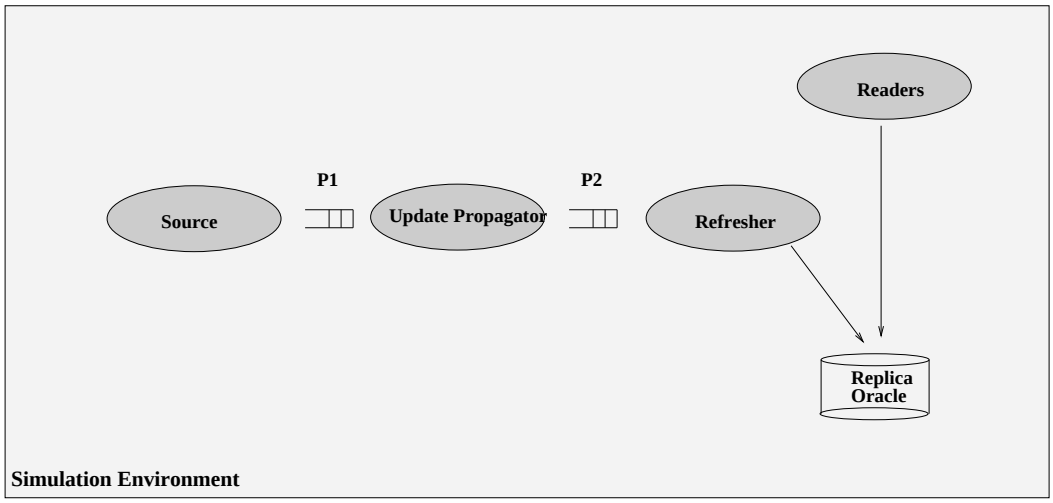

Figure 9: Simulator

to the sever for execution. The refresher process and the server are hosted in the same machine to avoid additional delays. The secondary copy uses a scheme definition based on AS3AP benchmark [Gra91] and was populated with 20000 tuples.

Finally, the third process, *readers*, implements query processing on the slave. The query expression we use is *Select atr from r where atr* $\geq c_1$ *and atr* $\leq c_2$, where c_1 and c_2 are fixed. Query arrival rate distribution (noted by λ_q) is defined to a load that we call *low*.

Using S2PL (strict two phase lock) as the underlying concurrency control protocol may increase query response time in conflict situations. A conflict situation happens when a query tries to read data items that are being updated. To improve query response time we consider the use of MV2PL (multiversion two phase lock) and examine its impact on freshness. MV2PL (multiversion two phase lock) explores the use of versions to increase concurrency between transactions. The principle is that queries reads committed versions of data items whereas update transactions write new ones. Its important to notice that queries never conflict with refresh transactions and do not need to take or wait for locks as with S2PL (strict two phase lock).

To compare the impacts of using S2PL and MV2PL we fix a 50% conflict situation. This means that the refresh transactions updates 50% of the tuples that are to be read by Q. We simulate S2PL by using the *select* command followed by *for update*. Since refresh transactions contains only write operations MV2PL is easily experienced by using Oracle multi-version consistency model [ea96] [1].

[1] Oracle provides "Snapshot Isolation" in which queries read old versions and update transactions write new versions through a technique called "READ CONSISTENCY".

Parameters	*Values*
ϵ	Exponential ($mean = 100ms$)
λ_t	Exponential: *low* ($mean = 10s$),*bursty* ($mean = 200ms$)
λ_q	Exponential: *low* ($mean = 15s$)
Q size	5
RT's sizes	5,50
Conflicts	50%
Protocols	S2PL, MV2PL
t_{short} (1 write)	300ms

Table 1: Performance Model

We define two update transactions types. When the update transaction size is 5 we call it a small transaction. In addition when update transaction size is 50 we call it long transaction. To understand the behavior of each strategy on the presence of short and long transactions we define four scenarios. Each scenario corresponds to a long transaction ratio (noted by ltr). In the first scenario, $ltr = 0$ (only short update transactions are executed), in the second $ltr = 30$ (30 % of the executed update transactions are long), in the third $ltr = 60$ (60 % of the executed update transactions are long) and in the forth $ltr = 100$ (all executed update transactions are long). The results we show are the averages values obtained for the execution of 40 update transactions.

Experiment 1

The goal of this experiment is to show the averages values of both freshness, query response time when using *deferred_immediate*, *immediate_immediate* and *immediate_wait* strategies for what we call a *low* update transaction arrival rate on the master.

Figure 10 shows that in average for the three strategies freshness is not significantly impacted. When $ltr = 0$ freshness is almost not impacted (0.3) because only short refresh transactions are produced and in this case, both propagation and refreshment times does not introduce a significant amount of delay if compared to λ_t. That is, in average, the interval of time between update transactions is big enough to permit update propagation and refreshment execution before another update transaction is triggered for execution. Therefore, deviations between primary an secondary copies are not so frequent. However, when long update transactions are introduced both propagation and refreshment time increases and impacts freshness. In these cases, refresh transaction are triggered later since propagation time increases. Furthermore, its execution time also increases due to the refresh transaction size. In this situation, *immediate_immediate* and *immediate_wait* improves freshness. However, the best results are obtained with *immediate_immediate*. Since we are mixing transaction sizes and our freshness measure is transaction based, freshness does not increases linearly with ltr.

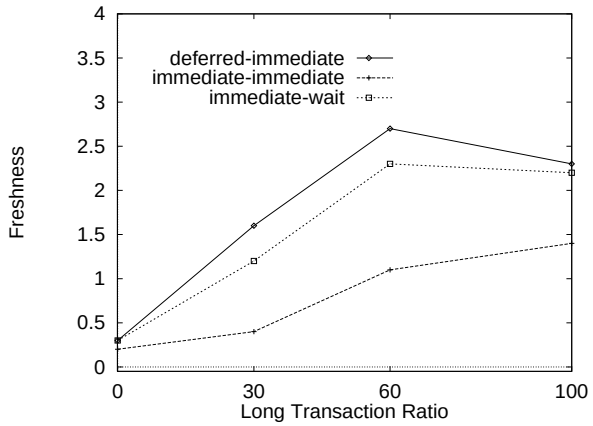

Figure 10: Low Workload - Freshness

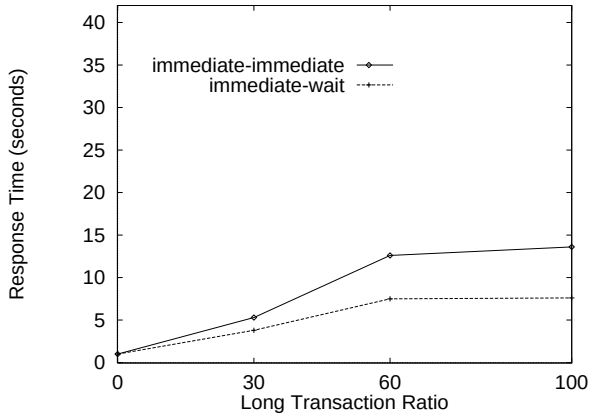

Figure 11: Low Workload - Response Time

Since the query size is small, the delays introduced when using S2PL in a conflict situation does not impact significatively refreshment time. However, it impacts queries response times. Figure 11 shows the response times. We do not show the curve for *deferred_immediate* since query response time is impacted in the same way when using *immediate_wait*. However, this not true with *immediate_immediate* since RT execution time is increases due to ϵ. During this period all data items that are being updated are *locked* until *commit* reception. As shown in Figure 11, with *immediate_immediate* in the presence of long transactions, query response times are almost doubled compared with *immediate_wait*.

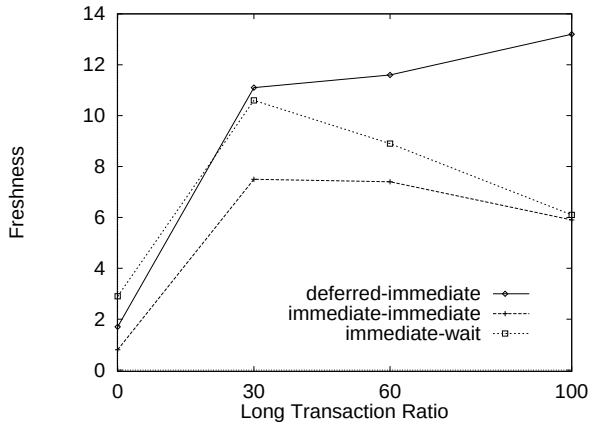

Figure 12: Bursty Workload - Freshness

Experiment 2

The goal of this experiment is to show the values of both freshness, query response times when using *deffered_immediate*, *immediate_immediate* and *immediate_wait* strategies for a high workload that we call *bursty*.

Figure 12 shows the freshness results. When $ltr = 0$ (only short transactions) freshness if not significantly impacted because the delay introduced by update propagation and refreshment is still not big enough compared to λ_t to cause a significant deviation between primary and secondary copies. It is important to notice that *deferred_immediate* presents better freshness results compared to *immediate_wait* in bursty workloads because δ increases sufficiently to impact update propagation time. Therefore, the time spent to propagate a short refresh transaction using *deferred* update propagation may be less than the time spent when using *immediate* update propagation. However, *immediate_immediate* still shows better results compared with both *deferred_immediate* and *immediate_wait* because the refresh transaction is triggered for execution as soon as the first write is received.

When long transactions are introduced freshness is significantly impacted because λ_t is small compared to the delay introduced by both update propagation and refreshment. Network contention may also occur, however the time spent to wait the complete execution of an update transaction before update propagation exceeds the contention delay. Notice that *immediate_wait* begins to improve better than *deferred_immediate* when ltr increases.

A queue of messages is produced in the slave. With *deffered_immediate* the queue is formed by *refresh transactions* messages. In addition, with immediate update propagation the queue is formed by *write* messages. For $ltr = 30$ and $ltr = 60$, *immediate_immediate* still presents better results compared with *immediate_wait* and *deffered_immediate*.

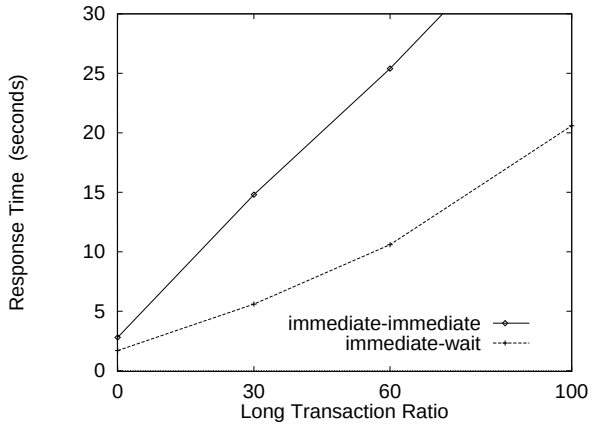

Figure 13: Bursty Workload - Response Time

For $ltr = 100$ (i.e. all update transactions are long and have the same size) *immediate_immediate* and *immediate_wait* presents almost the same freshness results due to the constant refresh transaction size. In this situation, freshness converges to a constant value for both *immediate_wait* and *immediate_immediate*. When using *immediate_wait* this value varies around the average value but not significantly. Figure 14 shows a freshness snapshot for a sequence of queries execution. Therefore, in this case, *immediate_wait* may be used in place of *immediate_immediate* with very small amount of loose of freshness. Notice that the gain of freshness when compared to *deferred_immediate* is more than a half. Besides, with *deferred_immediate* freshness does not converge to a constant value, instead, it keeps on growing.

Figure 13 shows the response times. When using *immediate_immediate* in the presence of long transactions, response times may be significantly delayed because of ϵ and network contention. Therefore, data items are blocked for longer periods of times compared to the low workload experiment. In this case, *immediate_wait* presents much better results.

4.3 Experiment 3

The goal of this experiment is to show the impacts of using MV2PL on freshness and response time. In average query response time for all cases (low and bursty workloads) is 1.2s. With respect to freshness Table 2 shows the results for the bursty workload. In general the loose of freshness is not impacting. The worst case occurs when $ltr = 30$ because refresh transactions have different sizes and the majority of the are small. The best case happens when $ltx = 100$ because transaction are all long and have the same size.

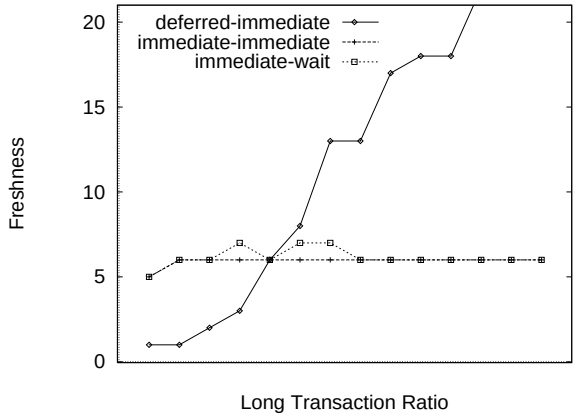

Figure 14: Bursty Workload - Freshness Behavior ($ltr = 100$)

ltr	*Deferred*	*Immediate-immediate*	*Immediate-wait*
0%	1.1	0.8	0.9
30%	1.7	2.8	1.3
60%	0.4	0.6	0.2
100%	1.7	0.3	0.2

Table 2: Bursty Workload - Freshness loose with MV2PL

Replication Scheme	*Ownership*	*Update Propagation*	*Refreshment*
A	Group	Deferred Immediate	Immediate (Reconciliation)
B	Master	Deferred Immediate	Immediate On Demand Group Periodic

Table 3: Replication Schemes

5 Related Work

Table 3 presents 2 lazy replication schemes and its basic parameters. We use this table to situate our work.

Replication scheme A corresponds to a lazy replication scheme where all replica copies are updatable (*update anywhere*). A *conflict* happens if two or more nodes update the same replicated object. There are several policies for conflict detection and resolution [Gol95, ea96] that can be based on timestamp ordering, node priority and others. The problem with conflict resolution is that during a certain period the database may be in a inconsistent state. Normally in this replication scheme uses the *deferred_immediate* strategy. However, the strategies proposed in this work may be used to detect and solve earlier conflict situations.

Replication scheme B is the focus of out work. There are several refreshment strategies for this replication scheme, since we are interested in freshness improvement we focus on deferred and immediate update propagation and refreshment strategies. With *on demand* refreshment, a replica copy is refreshed before a query is submitted for processing. Therefore, a delay may be introduced on query processing because all refresh transactions that were already received must be executed before processing the query. When *group* refreshment is used, refresh transactions are grouped executed in accordance to the applications freshness requirements. With *periodic* the approach, refreshment is triggered in fix interval of times. At refreshment time, all received refresh transactions are executed. In all cases, immediate update propagation may be experienced to improve freshness and optimize refreshment time.

[CMAP95] proposes a strategy called *incremental agreement* that has some features in common with our proposed strategies. However, they focus in managing network failures in replicated database and do not address the problem of improving freshness. Furthermore, optimization techniques are not proposed.

[GHOS96] compares the stability and convergence of replication schemes A and B through an analytical model. They introduce several concepts that were used in our work and explore the use of mobile and base nodes.

[GN95] presents formal concepts for specifying coherency conditions for replication scheme B in a large scale systems. These concepts permit to calculate an independent measure of

relaxation, called *coherency index*. In this context, versions conditions are closely related to our freshness measure.

Freshness measures are closely related to coherency conditions that are widely explored in [ABGM88, ABGM90, AA95, BGM90, SMAS94, WQ90, AKGM96, ZGMW95, WQ90, AKGM96].

[AKGM96] proposes several derived data refresh strategies: no batching, on demand, periodic and others for a similar scenario. Replica refreshment and derived data refreshment are done in separate transactions. They address freshness improvement, however they focus in the incoherency between derived data and the secondary copy.

6 Conclusions

In this paper, we presented a new frame work for lazy replication and proposed two strategies to improve freshness in a lazy master replication scheme. There behavior has been analyzed through practical experimentations and reveals that *immediate_immediate* strategy always improves freshness if compared with *deferred_immediate* and *immediate_wait*. The impacts are most significant for a special type of workload that we call *bursty*, specially when the majority of the update transactions are long. On the other hand, *immediate_wait only* shows results close to the ones revealed by *immediate_immediate* for *bursty* workload in the case where the majority of transactions are long. Using *immediate_wait* in these scenario avoids aborting transactions in case of network and site failures. Besides, refresh transaction and view maintenance optimization is possible, permitting even more the increase of freshness.

The downside of using *immediate_immediate* is the increase of query response time due to network delays. However, query response time may be reduced by using *immediate_wait*. Furthermore, it can be drastically reduced by using a multiversion protocol without a signicant loose of freshness, as shown in our experimentations. Finally, our analysis reveals that network traffic has significant impact on *immediate_wait*, specially if the majority of the transactions are small.

7 Acknowledgments

We would like to thank Dennis Shasha for giving us the motivation of this work and all his generous attention during its evolution, Anthony Tomasic for his clever comments on the key points of the work and the encouragement for its concretization. We would also like to thank Francoise Fabret for her precise comments on earlier drafts of this paper and Patrick Valduriez for the final comments. Many thanks to Luc Bouganim, Hubert Naacke and Florian Xhumari for all implementation hints.

References

[AA95] G. Alonso and A. Abbadi. Partitioned data objects in distriduted databases. *Distributed and Parallel Databases*, (3):5–35, 1995.

[ABGM88] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Quasi-copies: Efficient data sharing from information retrieval systems. *Proceedings of the Int. Conf. on Extending Data Base Technology (EDBT)*, pages 373–387, 1988.

[ABGM90] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, (359-384), 1990.

[AKGM96] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. *Database Support for Efficiently Maintaining Derived Data.* 1996.

[BGM90] Daniel Barbara and Hector Garcia-Molina. The case controlled inconsistency in replicated data. *Proceedings of the Workshop on Management of Replicated Data*, pages 35 –38, 1990.

[CFLS91] M.J. Carey, M.J. Franklin, M. Livny, and E.J. Shekita. Data caching tradeoffs in client-server DBMS architectures. *Proceedings of the ACM SIGMOD*, pages 357–366, June 1991.

[CMAP95] S. Cerri, M.A.W.Houstma, A.M.Keller, and P.Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, July 1995.

[Dav94] Judith R. Davis. Data replication. *Distributed Computing Monitor*, 1994.

[ea96] Steven Bobrowski et. al. Oracle 7 server concepts, release 7.3. *Oracle Corporation, 500 Oracle Parkway, Redwood City, CA*, 1996.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The danger of replication and a solution. *Proceedings ACM SIGMOD*, pages 173–182, June 1996.

[GN95] Rainer Gallersdorfer and Matthias Nicola. Improving performance in replicated databases through relaxed coherency. In *Proceedings of the 21st VLDB Conference*, pages 445 – 456, 1995.

[Gol95] Rob Goldring. Things every update replication customer should know. *Proceeding of the ACM Sigmod*, (439-440), June 1995.

[GR92] J.N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, August 1992.

[Gra91] *The Benchmark Handbook.* Morgan Kaufmann, jim gray edition, 1991.

[KR87] Bo Kahler and Oddvar Risnes. Extending logging for database snapshot refresh. *Proceedings of the 13th VLDB Conference, Brighton*, pages 389–398, 1987.

[Lad90] Rivka Ladin. Lazy replication: Exploiting the semantics of distributed services. *Proceedings of the Workshop on Management of Replicated Data*, pages 31–34, 1990.

[Moi96] Alex Moissis. *Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information*, 1996.

[PHN87] P.A.Bernstein, V. Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems.* 1987.

[SKS86] Sunil K. Sarin, Charles W. Kaufman, and Janet E. Somers. Using history information to process delayed database updates. *Proceedings of the 12th International Conference on VLDB*, pages 71–78, 1986.

[SMAS94] S.Ceri, M.A.W.Houtsma, A.M.Keller, and P. Samarati. A classification of update methods for replicated databases. Technical report, Standford University, 1994.

[WQ90] Gio Wiederhold and Xiaolei Qian. Consistency control of replicated data in federated databases. *Proceedings of the Workshop on Management of Replicated Data*, 1990.

[ZGMW95] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. View maintenance in a warehouse environment. *Proceedings ACM SIGMOD*, pages 316–327, 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399